Capítulo 11 - SENSEIGMNS

El modelo de sincronía virtual define un servicio GMS de pertenencia a grupos cuya funcionalidad se basa en dos operaciones: inclusión y exclusión de un miembro en un grupo. SenseiGMS no define ninguna operación pública para la creación de grupos, y esta tarea se delega a SenseiGMNS, que constituye, consecuentemente, una parte integral de Sensei.

El servicio fundamental que SenseiGMNS ofrece es la gestión básica de grupos, permitiendo crearlos o extenderlos, supliendo así la funcionalidad no presente en SenseiGMS. Al emplear este servicio, la aplicación debe decidir si desea crear o extender un grupo, y en este último caso, tiene que suministrar un miembro del grupo en el que quiere incluirse.

Para facilitar esta gestión de grupos, SenseiGMNS implementa también un servicio de directorio, permitiendo manejar los grupos en base a los nombres que se les asigna. En este caso, una aplicación suministra exclusivamente el nombre del grupo en el que quiere incluirse y el servicio decide, en base a la información que mantiene, si se trata de un grupo nuevo que debe crearse o de un grupo que debe extenderse. Este servicio da nombre a SenseiGMNS, acrónimo de las siglas en inglés de *Group Membership Naming Service*.

Además de describir la funcionalidad de SenseiGMNS, este capítulo muestra el empleo de la metodología soportada por SenseiDomains, pues la implementación se realiza empleando componentes dinámicos.

SenseiDomains se diseñó con independencia del substrato de comunicaciones empleado. SenseiGMNS, sin embargo, está ligado a SenseiGMS y no tiene sentido sin éste. Otros sistemas de comunicaciones en grupo implementan ambas funcionalidades en un solo servicio, por lo que es innecesario el empleo de SenseiGMNS en el caso de utilizar SenseiDomains sobre estos otros sistemas.

11.1. Gestión básica de grupos

Un sistema de comunicaciones en grupo fiables debe soportar la gestión de grupos de réplicas, permitiendo a un servidor crear un nuevo grupo, extenderlo, o excluirse del mismo. El modelo de sincronía virtual no especifica cómo debe realizarse esa gestión; *Ensemble*, por ejemplo, la realiza identificando cada grupo con un nombre y empleando este nombre en la operaciones de creación y extensión de grupos (ambas funciones las implementa con una única función *join*).

La interfaz pública de SenseiGMS no incluye ninguna operación con este propósito, aunque sí se incluye en la interfaz privada, sobre *SenseiGMSMember*:

```
interface SenseiGMSMember : GroupHandler
{
        boolean createGroup();
        boolean addGroupMember(in SenseiGMSMember other);
        boolean joinGroup(in SenseiGMSMember group);
        . . .
};
```

El enfoque tomado es interpretar cada miembro de un grupo como un punto de acceso al grupo. Una réplica debe obtener de alguna manera una referencia a otra réplica para incorporarse al grupo al que esta última pertenece. No es un problema que una réplica pertenezca a varios grupos pues, en ese caso, tiene diferentes referencias *SenseiGMSMember* para cada grupo.

Para que un servidor pueda pertenecer a un grupo, debe implementar la interfaz *GroupMember*; al incorporarse al grupo, obtiene una referencia *GroupHandler* con la que accede a las operaciones en grupo. La relación entre interfaces se muestra en la figura 11.1:

- El servidor implementa la interfaz *GroupMember*, mediante la cual obtiene los eventos de grupo.
- El servidor obtiene una referencia *GroupHandler*, con la que realiza las comunicaciones al grupo.
- La interfaz *GroupHandler* obtenida es una instancia de *SenseiGMSMember*. El anillo de comunicaciones del grupo en Sensei se forma con las instancias *SenseiGMSMember* de sus miembros.

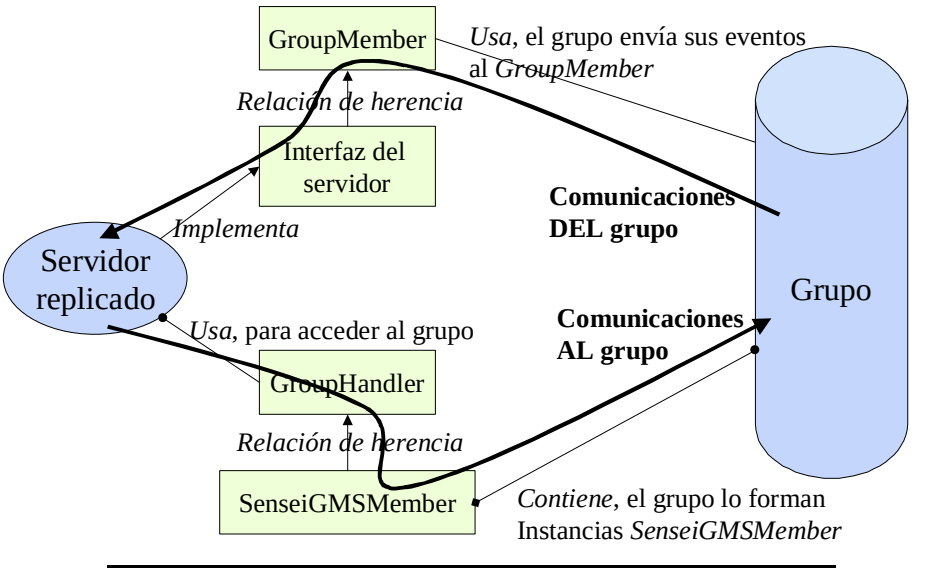

Figura 11.1. Relación de interfaces en *SenseiGMS*

Por lo tanto, todo miembro del grupo debe instanciar la interfaz *SenseiGMSMember*, con la que pasa a formar parte del anillo de comunicaciones de su grupo. Es perfectamente posible que un servidor en una máquina determinada emplee una instancia *SenseiGMSMember* localizada en una máquina diferente, pero introduce un nuevo punto de error, reduciendo la tolerancia a fallos. Por ejemplo, la figura 11.2 visualiza un escenario donde una aplicación replicada con dos servidores mantiene las instancias *SenseiGMSMember* en la misma máquina. Si esta máquina se cae, no sólo se cae el servidor que alberga, sino que el otro servidor pierde igualmente su funcionalidad. La aplicación no era tolerante a fallos en la máquina caída, lo que ha supuesto su pérdida de servicio.

El razonamiento realizado implica que cuando un servidor solicita su inclusión a otro servidor de un grupo dado, éste debe crear una instancia *SenseiGMSMember* que resida en la máquina y proceso del primer servidor. En lugar de extender la interfaz *GroupMember* con una operación *callback* que realice este cometido, SenseiGMNS define una interfaz específica *GroupHandlerFactory*:

```
exception GroupHandlerFactoryException{};
interface GroupHandlerFactory
{
       GroupHandler create() raises (GroupHandlerFactoryException);
};
```

Un servidor que accede a un miembro para unirse a su grupo, suministra una instancia *GroupHandlerFactory* que el miembro destino emplea para crear la instancia *GroupHandler*, la cual resulta ahora local al nuevo miembro.

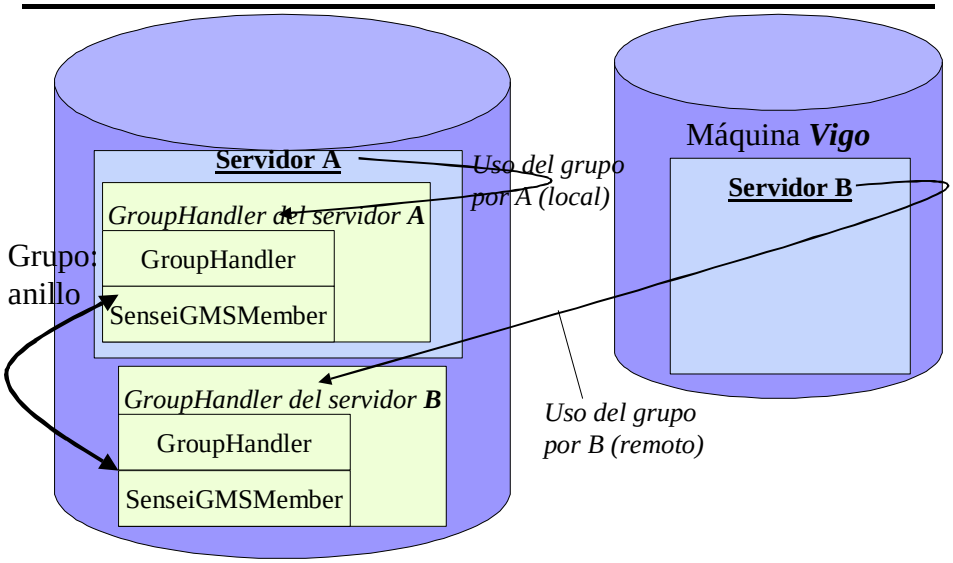

Figura 11.2. Creación errónea de instancias *GroupHandler*

Esta lógica supone que toda aplicación tolerante a fallos debe ser capaz de crear un objeto de tipo *SenseiGMSMember* y de implementar, consecuentemente, los algoritmos de comunicaciones en grupo desarrollados en el capítulo 8, cuando explicamos SenseiGMS. Una alternativa es emplear la funcionalidad de CORBA de transferencia de objetos por valor para este fin: el servidor GMNS puede suministrar el objeto con funcionalidad *SenseiGMSMember*, para lo que definimos la siguiente entidad:

```
valuetype GroupHandlerFactoryCreator
{
       GroupHandlerFactory create(in GroupMember member)
              raises (GroupHandlerFactoryException);
};
```

De esta manera, un miembro accede al servicio SenseiGMNS, obteniendo un objeto *GroupHandlerFactoryCreator*. Su definición es *valuetype* en lugar de *interface*, lo que garantiza que esta referencia *GroupHandlerFactoryCreator* reside localmente en el espacio de memoria del miembro que la solicita[10]. A continuación, este miembro crea un objeto *GroupHandlerFactory* suministrando una referencia a su

[10] Esta funcionalidad está directamente soportada en Java, que permite transferencia no sólo de datos, sino también de código. Así, la interfaz JavaRMI de *SenseiGMNS* define el tipo *GroupHandlerFactoryCreator* como una extensión de *java.util.Serializable*, mientras que todo tipo remoto extiende *java.rmi.Remote*.

interfaz *GroupMember*, y ese objeto será empleado en las operaciones de creación o extensión del grupo.

La interfaz básica de SenseiGMNS es *GroupMembershipBasicService*, definida en `GroupMembershipNamingService.idl` de la siguiente forma:

```
exception InvalidGroupHandlerException{};
exception InvalidGroupHandlerFactoryException{};
interface GroupMembershipBasicService
{
        GroupHandler createGroup(in GroupHandlerFactory theFactory)
                raises (InvalidGroupHandlerFactoryException,
                        GroupHandlerFactoryException);
        GroupHandler joinGroup(in GroupHandler group,
                        in GroupHandlerFactory theFactory)
                raises (InvalidGroupHandlerException,
                        InvalidGroupHandlerFactoryException,
                        GroupHandlerFactoryException);
        GroupHandlerFactoryCreator getFactoryCreator();
};
```

- *createGroup*: crea un grupo a partir de un *GroupHandlerFactory*. La instancia *GroupHandler* que se devuelve se crea de forma local en el miembro que invoca la operación, y no local al servicio GMNS.
- *joinGroup*: extiende el grupo identificado por la referencia *GroupHandler* dada con un nuevo miembro, especificado mediante la referencia *GroupHandlerFactory*.
- *getFactoryCreator*: devuelve la instancia *GroupHandlerFactoryCreator*, con la que es posible crear los objetos *GroupHandlerFactory* empleados en las dos operaciones anteriores.

El escenario asociado resulta complejo para asegurar que el anillo lógico de comunicaciones de Sensei se integra por componentes residentes físicamente en la máquina y proceso que representan. Resumiendo, los pasos requeridos para incluir un miembro en un grupo son:

- El futuro miembro del grupo accede al servicio GMNS y obtiene mediante la operación *getFactoryCreator* un objeto del tipo *GroupHandlerFactoryCreator*, que reside localmente en la máquina y proceso de ese miembro.
- Este miembro emplea el objeto *GroupHandlerFactoryCreator* para crear una instancia de la interfaz *GroupHandlerFactory* que existe, por lo tanto, localmente.
- El miembro accede de nuevo al servicio GMNS, esta vez a la operación *joinGroup*, pasando el objeto *GroupHandlerFactory*.
- El servidor GMNS accede (remotamente) al objeto *GroupHandlerFactory*, para crear el *GroupHandler* en el proceso del futuro miembro, y devuelve la referencia creada a ese miembro, ya perteneciente al grupo.

La interfaz del servicio creado para gestionar los grupos no ofrece, sin embargo, ninguna funcionalidad para obtener referencias a otros miembros, que deben ser localizadas por la aplicación. El servicio de directorio de SenseiGMNS cubre esta funcionalidad.

11.2. Servicio de directorio

El servicio de directorio mantiene referencias a los miembros existentes en un grupo, al que identifica mediante un nombre. Empleando esta funcionalidad, un miembro puede incluirse en un grupo especificando su nombre y el servicio se encarga de crear un nuevo grupo o, si ya contiene referencias a otros miembros en el mismo grupo, de extenderlo.

La utilidad de este servicio no se limita a la gestión de grupos. Un cliente puede emplearlo para obtener la referencia de un miembro de un servidor replicado, a diferencia del servicio mostrado en la anterior sección.

Desde la perspectiva del grupo, cada miembro debe implementar la interfaz *GroupMember*. Pero estos miembros forman, además, parte de una aplicación que ofrece una interfaz específica a sus clientes. El servicio de directorio almacena referencias de los miembros a ambas interfaces y, por ello, se define la interfaz *ReplicatedServer*, sin funcionalidad propia, que debe ser extendida por la interfaz de cliente:

```
interface ReplicatedServer
{
};
```

El servicio de directorio se define mediante la siguiente interfaz:

```
interface GroupMembershipNamingService
{
        GroupHandlerFactoryCreator getFactoryCreator();
        GroupHandler findAndJoinGroup(in string groupName,
                        in GroupHandlerFactory theFactory,
                        in string memberName,
                        in ReplicatedServer clientsReference)
                raises (sensei::middleware::domains::MemberStateException,
                        InvalidGroupHandlerFactoryException,
                        GroupHandlerFactoryException);
        void leaveGroup(in string groupName, in GroupHandler member);
        ReplicatedServersList getGroup(in string groupName)
                raises (sensei::middleware::domains::MemberStateException);
        ReplicatedServer getGroupServer(in string groupName, in string memberName)
                raises(sensei::middleware::domains::MemberStateException);
```

```
        ReplicatedServer getValidGroupServer(in string groupName)
                raises(sensei::middleware::domains::MemberStateException);
};
```

- *getFactoryCreator*: define la misma funcionalidad que en el caso de la operación homónima en *GroupMembershipBasicService*.
- *findAndJoinGroup*: crea o extiende un grupo, al que se identifica por un nombre. El miembro debe suministrar un nombre, una interfaz de cliente y una instancia *GroupHandlerFactory*, usando la misma filosofía mostrada en *GroupMembershipBasicService* para crear los objetos localmente al miembro que extiende o crea el anillo.
- *leaveGroup:* el miembro identificado por la instancia *GroupHandler* asociada es expulsado del grupo.
- *getGroup*: devuelve las referencias de todas las réplicas del grupo especificado, sin comprobarse si las réplicas son aún válidas. Las referencias devueltas son las interfaces de cliente registradas.
- *getGroupServer*: devuelve la referencia a un servidor específico en el grupo dado. La referencia devuelta es la interfaz de cliente registrada.
- *getValidGroupServer*: devuelve la referencia a un servidor específico en el grupo dado, comprobándose previamente que el servidor es todavía válido. La referencia devuelta es la interfaz de cliente registrada.

El servicio GMNS comprueba periódicamente las réplicas que contiene, eliminando aquellas que han caído o hayan sido excluidas del grupo sin haber empleado *leaveGroup*.

Si el servicio básico de SenseiGMNS es esencial para la creación de grupos, este servicio de directorio no lo es. Así, un miembro que se incluye en un grupo sin emplear este servicio de directorio no es registrado y no puede ser utilizado para extender el grupo al que pertenece. Las consecuencias son posibles inconsistencias en aplicaciones que no empleen de forma uniforme el servicio básico o el servicio de directorios de SenseiGMNS.

Para que este servicio sea efectivo, debe ser por sí mismo tolerante a fallos por lo que se ha diseñado como un servidor replicado, diseño que detallamos a continuación como ejemplo de uso de la metodología que SenseiDomains implementa.

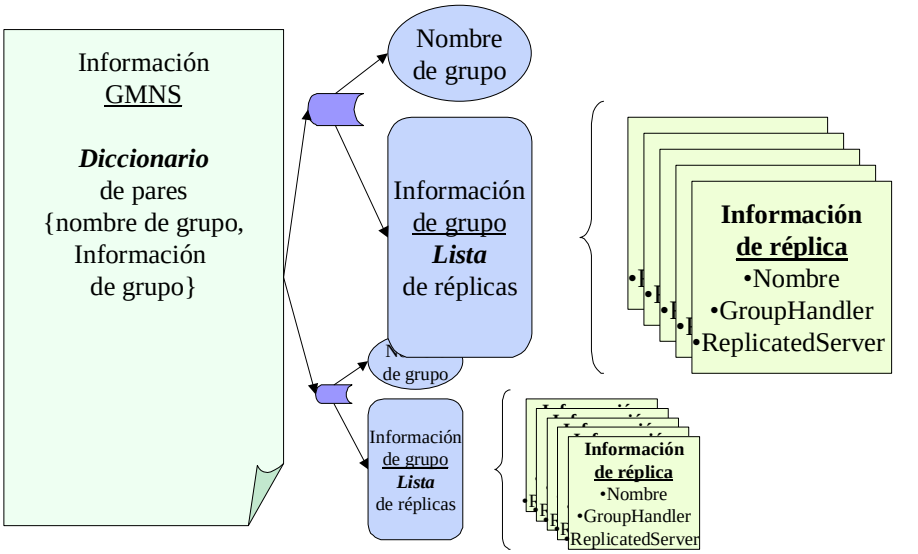

Figura 11.3. Estructura de datos en un servidor GMNS no replicado

11.3. Diseño de SenseiGMNS en componentes dinámicos

La sección 8.3 introdujo las bases de diseño de un servicio de directorio como ejemplo de uso de SenseiGMS. En aquel caso, el diseño se basaba en un único componente *GMNSdata* que almacenaba la información a replicar en el servicio. Este componente no soportaba accesos concurrentes, con lo que varias réplicas no podían actualizarlo simultáneamente.

El diseño real de un servidor GMNS replicado lo hemos realizado extendiendo el modelo del mismo servidor no replicado. Sobre este servicio no tolerante a fallos, una implementación lógica se basaría [figura 11.3] en un contenedor tipo *diccionario* que contendría como claves los nombres de los grupos y como valores la información asociada a éstos. La información necesaria para un grupo puede, a su vez, representarse con un contenedor tipo *lista* donde cada elemento es la información de una réplica, compuesta por sus referencias *GroupHandler* y *ReplicatedServer*, y el nombre asignado.

Con esta estructura de datos, el diccionario estaría inicialmente vacío; al crearse un nuevo grupo, se crea una nueva instancia del tipo lista empleado para almacenar la información de las réplicas de un grupo, y se introduce un par {nombre, lista} en el diccionario. A continuación, se crea una instancia de la clase empleada para almacenar la información de una réplica, que se inserta en la lista

dada. Por consiguiente, el diccionario es el único componente inicial, todos los demás se crean dinámicamente.

Al transformar el servidor en uno replicado bajo CORBA, sería necesario definir primero la entidad que almacena la información de una réplica, para lo que empleamos la *struct MemberInfo*:

```
struct MemberInfo
{
        GroupHandler handler;
        string name;
        ReplicatedServer publicReference;
};
typedef sequence<MemberInfo> MemberInfoList;
```

El contenedor que guarda las instancias *MemberInfo* es una lista, por lo que se podría implementar con el contenedor replicado estándar *ReplicatedList* de SenseiUMA. Sin embargo, *SenseiUMA*, a diferencia de los demás sistemas en *Sensei* se define de forma diferente en CORBA y JavaRMI, pues en su implementación bajo JavaRMI intenta integrarse con tipos ya definidos, como *java.util.iterator*, etc. Por esta razón, SenseiGMNS se implementaría también diferentemente en ambas arquitecturas, en contra de lo deseado. Además, implicaría una dependencia de SenseiGMNS en SenseiUMA, dependencia que preferimos evitar para minimizar el código estrictamente necesario en la implementación de SenseiGMNS.

Como consecuencia, definimos un contenedor del tipo *lista replicada* especialmente para este caso. El principal beneficio asociado es que en lugar de emplear tipos genéricos, la lista se define como un contenedor de instancias concretas *MemberInfo*:

```
interface SetMemberInfo : ExtendedCheckpointable, SetMemberInfoObservable
{
        boolean add(in MemberInfo member)
                raises (sensei::middleware::domains::MemberStateException);
        boolean remove(in MemberInfo handler)
                raises (sensei::middleware::domains::MemberStateException);
        MemberInfo get()
                raises (sensei::middleware::domains::MemberStateException);
        MemberInfoList toArray()
                raises (sensei::middleware::domains::MemberStateException);
};
```

El tipo de contenedor es *set*, una lista que no acepta valores duplicados. Puesto que este contenedor no tiene un uso general, se definen sólo las operaciones requeridas: insertar y borrar un elemento, que devuelven un valor booleano que indica el éxito de la aplicación, y dos operaciones de acceso a los elementos contenidos. Este acceso es también específico y, en lugar de definirse una interfaz

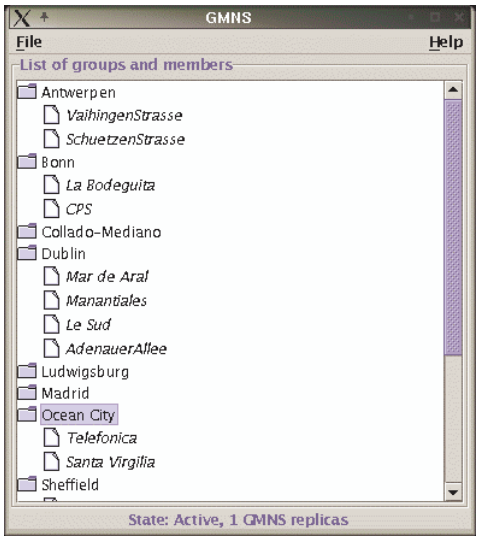

Figura 11.4. Interfaz gráfica de *SenseiGMNS*

que itere sobre el contenido, al estilo de *java.util.Iterator* en Java, se emplean dos operaciones básicas: obtener un elemento y obtener todos los elementos.

La definición de esta interfaz es *ExtendedCheckpointable*, luego el soporte de transferencia de estado es básico y la transferencia se efectúa en un solo paso. Además, la interfaz extiende *SetMemberInfoObservable*, que se define como:

```
interface SetMemberInfoObservable
{
       void addObserver(in SetMemberInfoObserver observer);
       boolean removeObserver(in SetMemberInfoObserver observer);
};
```

La funcionalidad asociada es permitir observar al contenedor dado. Puesto que este contenedor está replicado, cualquiera de los servidores GMNS puede actualizarlo, y esas actualizaciones deben reflejarse en la interfaz gráfica del servicio [figura 11.4], lo que admite dos soluciones: interrogar periódicamente a los contenedores sobre su contenido o recibir eventos cuando se actualizan. La segunda aproximación es más complicada, pero ventajosa. De hecho, SenseiUMA soporta también esta segunda opción en los contenedores definidos.

La observación se realiza aplicando el patrón de observación [Gamma95], tambien conocido como patrón de publicación/subscripción: el contenedor se define como *observable*, permitiendo que todo elemento registrado, implementando una interfaz *Observer*, reciba los eventos de cambios. Esta última interfaz se define para este caso específico como:

```
interface SetMemberInfoObserver
{
        void addDone(in SetMemberInfoObservable observable, in MemberInfo member);
        void removeDone(in SetMemberInfoObservable observable, in MemberInfo member);
};
```

La lista informa, por lo tanto, de cualquier adición o borrado de elementos, junto con el elemento dado.

La misma lógica general se emplea para el diccionario, que se define mediante la interfaz *MapStringSet*:

```
exception InvalidSetMemberInfo{};
typedef sequence<string> stringList;

interface MapStringSetObserver
{
        void clearDone(in MapStringSetObservable observable);
        void putDone(in MapStringSetObservable observable,
                        in string groupName,
                        in SetMemberInfo set);
        void removeDone(in MapStringSetObservable observable,
                        in string groupName,
                        in SetMemberInfo set);
};

interface MapStringSetObservable
{
        void addObserver(in MapStringSetObserver observer);
        boolean removeObserver(in MapStringSetObserver observer);
};

interface MapStringSet : ExtendedCheckpointable, MapStringSetObservable
{
        SetMemberInfo put(in string groupName, in SetMemberInfo set)
                raises (InvalidSetMemberInfo,
                        sensei::middleware::domains::MemberStateException);
        SetMemberInfo remove(in string groupName)
                raises (sensei::middleware::domains::MemberStateException);
        SetMemberInfo get(in string groupName)
                raises (sensei::middleware::domains::MemberStateException);
        stringList getKeys()
                raises (sensei::middleware::domains::MemberStateException);
};
```

11.3.1. Implementación de los componentes

Tras haber definido la interfaz de los componentes de SenseiGMNS, es necesario especificar los mensajes que las réplicas intercambian y cómo se transfiere el estado.

MemberInfo se definió en OMG/IDL como *struct*: no tiene comportamiento, sólo almacena los datos asociados. En el caso del componente *SetMemberInfo,* soporta dos operaciones: *add*, *remove*. En lugar de definir dos mensajes, empleamos uno solo, especificado como:

```
valuetype SetMemberInfoChangedMessage : DomainMessage
{
       public MemberInfo info;
       public boolean add;
};
```

Este mensaje contiene la información del miembro que se añade o elimina, y se define como una especialización de *DomainMessage*. No precisa de un tratamiento especial durante las transferencias o transacciones, por lo que la implementación no debe modificar el valor del campo *unqueuedOnST* (*false*) de *DomainMessage*.

El componente se define como *ExtendedCheckpointable*, luego es estado se transfiere en un único paso; en este caso, la información a transferir incluye los datos de cada uno de los componentes de la lista:

```
valuetype SetMemberInfoState : State
{
       public sequence<MemberInfo> members;
};
```

La implementación del componente la realizamos en Java; la clase asociada necesita dos constructores distintos:

```
SetMemberInfoImpl(int subgroupId, DomainGroupHandler groupHandler);
SetMemberInfoImpl(DynamicSubgroupInfo dynamicInfo, DomainGroupHandler groupHandler);
```

- Constructor *estático*. Empleado en caso de registrar el componente estáticamente en el dominio, precisa de una identidad de componente predefinida y del dominio a emplear. También se utiliza para crear componentes dinámicos en demanda, al conocerse ya su identidad.
- Constructor *dinámico*. Empleado para crear dinámicamente un componente, precisa de la información a enviar a otras réplicas.

El componente implementa una lista, por lo que emplea internamente una lista para almacenar los miembros; el tipo seleccionado es *java.util.Set* (implementado mediante *java.util.HashSet*). El estado inicial de esta lista no incluye ningún miembro, y se implementan las operaciones de transferencia como:

```
public synchronized State getState()
{
        return factory.createSetState(getMembers());
}

MemberInfo[] getMembers()
{
        int size = set.size();
        MemberInfo members[] = new MemberInfo[size];
        if (size>0) {
                int i=0;
                Iterator it = set.iterator();
                while(it.hasNext()) {
                        members[i++]=(MemberInfo) it.next();
              }
        }
        return members;
}

public synchronized void setState(State state)
{
        set.clear();
        MemberInfo members[]=((SetMemberInfoState)state).members;
        int size = members.length;
        for (int i=0;i<size;i++) {
                addMemberInfo(members[i]);
        }
}

synchronized boolean addMemberInfo(MemberInfo memberInfo)
{
        boolean ret = set.add(memberInfo);
        if (ret) {
                observersHandler.informPut(memberInfo);
        }
        return ret;
}

public synchronized void assumeState()
{
        set.clear();
}
```

- *assumeState*: elimina cualquier elemento de la lista.

- *setState*: añade cada uno de los elementos dados en la lista de entrada. Para ello emplea un método *addMemberInfo*, también empleado en la adición posterior de miembros que, además de incluirlo en la lista, informa del evento a los posibles observadores.
- *getState*: emplea un objeto *factory*, capaz de crear el objeto *middleware State*, tanto para JavaRMI, como para CORBA, con la información del miembro. Esta información la obtiene de *getMembers*, que itera sobre los elementos de la lista, devolviéndolos en un *array*.

El componente *SetMemberInfo* es un miembro de grupo, por lo que debe reaccionar a los seis eventos del grupo. Sólo es relevante el procesado de los mensajes multipunto:

```
public void processPTPMessage(int parm1, Message parm2){}
public void changingView(){}
public void installView(View parm1){}
public void memberAccepted(int id, GroupHandler parm2, View parm3)
{
        memberId=id;
}
public void excludedFromGroup()
{
        freeResources();
}
public void processCastMessage(int sender, Message message)
{
        if (message instanceof SetMemberInfoChangedMessage) {
                synchronized(this) {
                        SetMemberInfoChangedMessage msg =
                                (SetMemberInfoChangedMessage) message;
                        if (msg.add)
                                addMemberInfo(msg.info);
                        else
                                removeMemberInfo(msg.info);
                }
        }
}
```

En caso de recibir un mensaje *SetMemberInfoChangedMessage*, se añade o elimina el miembro que contiene, según sea el mensaje. Para ello, se emplea el método *addMemberInfo*, ya mostrado anteriormente, o *removeMemberInfo* que, de la misma manera, elimina el miembro de la lista e informa a los observadores:

```
synchronized boolean removeMemberInfo(MemberInfo memberInfo)
{
        Iterator it = set.iterator();
```

```
        while(it.hasNext()) {
                MemberInfo content = (MemberInfo) it.next();
                if (ObjectsHandling.areEquivalent(memberInfo.handler, content.handler))
                {
                        it.remove();
                        observersHandler.informRemove(memberInfo);
                        return true;
               }
        }
        return false;
}
```

En CORBA, dos referencias a un mismo miembro pueden ser distintas, por lo que es necesario emplear métodos CORBA específicos para realizar la comparación entre referencias. El objeto *ObjectsHandling* efectúa esta comparación de modo transparente para CORBA y JavaRMI, permitiendo que el mismo código sea válido en ambas plataformas.

La interfaz pública del componente se implementa a continuación, sin incluir el tratamiento de errores. El código es una implementación directa del patrón de sincronización de mensajes en el caso de tener que devolver un valor al cliente:

```
public boolean add(MemberInfo info) throws MemberStateException
{
        boolean result=false;
        factory.castSetMessage(info, true);
        result=addMemberInfo(info);
        factory.syncMessageCompleted();
        return result;
}
public synchronized MemberInfo get() throws MemberStateException
{
        Iterator it=set.iterator();
        return it.hasNext()? (MemberInfo) it.next(): null;
}
public boolean remove(MemberInfo info) throws MemberStateException
{
        boolean result=false;
        factory.castSetMessage(info, false);
        result=removeMemberInfo(info);
        factory.syncMessageCompleted();
        return result;
}
public synchronized MemberInfo[] toArray()
{
        int size = set.size();
```

```
        MemberInfo[] ret = new MemberInfo[size];
        set.toArray(ret);
        return ret;
}
```

Por consiguiente, las operaciones *get* y *toArray* se implementan sin comunicaciones al grupo, y las operaciones *add* y *remove* implican el envío de un mensaje y su procesado en el mismo *thread* de envío, devolviendo el resultado de la operación al cliente. El código necesario para implementar el patrón de observación es irrelevante y no lo incluimos para no extender innecesariamente esta explicación; el apéndice D contiene el código completo de esta clase.

El componente *MapStringSet* sigue la misma lógica. Define los mismos constructores, mismo tratamiento de eventos, etc. A continuación se muestra la definición de los mensajes y el estado:

```
valuetype MapStringSetPutMessage : DomainMessage
{
        public string key;
        public SubgroupId value;
};

valuetype MapStringSetRemoveMessage : DomainMessage
{
        public string key;
};

struct MapStringSetEntry
{
        string key;
        SubgroupId value;
};

valuetype MapStringSetState : State
{
        public sequence<MapStringSetEntry> groups;
};
```

En este caso se definen dos mensajes distintos para las operaciones *put* y *remove*, pues la información asociada es diferente.

Es importante destacar que en las comunicaciones al grupo no se envía el componente *SetMemberInfo* que contiene, sino la identidad de ese componente replicado (en negrita en el anterior listado), tal como se ha registrado en el dominio. Este tratamiento no es una optimización, sino una necesidad, pues cada réplica debe contener referencias de los componentes que instancia, no de los componentes en la réplica que envía la información.

Ésta es precisamente la segunda razón por la que SenseiUMA es diferente en JavaRMI y CORBA. Java soporta nativamente patrones de serialización y es posible especificar la manera en que un determinado objeto se serializa. Al serializar objetos del tipo *GroupMember*, se accede al dominio para obtener su identidad de componente, que es la información que se envía. Bajo CORBA, el componente podría verificar explícitamente si el objeto a enviar implementa la interfaz *GroupMember*, pero esta solución resulta insuficiente, al no cubrir objetos complejos que incluyan en su composición componentes replicados.

Este componente implementa un diccionario o *mapa*, por lo que emplea internamente el mismo tipo para almacenar elementos; *java.util.Map* (implementado mediante *java.util.HashMap*). El apéndice D incluye el código completo de esta clase, y sólo mostramos aquí el tratamiento de componentes replicados en el método *put*, que incluye para un nombre de grupo un componente *SetMemberInfo*, y en el método de recepción de mensajes, *processCastMessage*, ambos sin comprobación exhaustiva de errores:

```
public SetMemberInfo put(String group, SetMemberInfo set)
        throws InvalidSetMemberInfo, MemberStateException
{
        SetMemberInfo result=null;
        int subgroupId = groupHandler.getSubgroupId(set);
        if (subgroupId==Consts.INVALID_SUBGROUP) {
                throw new InvalidSetMemberInfo();
        }
        factory.castMapPutMessage(group, subgroupId);
        result = doPut(group, set);
        factory.syncMessageCompleted();
        return result;
}

public void processCastMessage(int sender, Message message)
{
        if (message instanceof MapStringSetPutMessage) {
                MapStringSetPutMessage msg = (MapStringSetPutMessage) message;
                GroupMember subgroup = groupHandler.getSubgroup(msg.value);
                SetMemberInfo set = GMNSnarrower.toSetMemberInfo(subgroup);
                if (set!=null) {
                        doPut(msg.key, set);
                }
        }
        . . .
}

SetMemberInfo doPut(String group, SetMemberInfo set)
{
```

```
        SetMemberInfo ret = null;
        Object obj = null;
        synchronized(this){obj=map.put(group, set);}
        if (obj!=null) {
                ret = GMNSnarrower.toSetMemberInfo(obj);
                observersHandler.informRemove(group, ret);
        }
        observersHandler.informPut(group, set);
        return ret;
}
```

- Al procesar una operación *put*, se comprueba que el componente insertado es válido (que esté registrado en el dominio). En la estructura de datos interna se guarda la referencia al componente replicado, aunque la información que se envía al grupo es su identidad.
- Al recibir el mensaje asociado a la operación *put*, se obtiene el componente replicado a partir de la identidad de componente recibida.
- Ambos métodos delegan en la operación *doPut* para insertar efectivamente el par {nombre, componente} que, además, comunica el evento a los observadores.

11.3.2. Integración de componentes

Al introducir información relativa a un nuevo grupo, el servidor GMNS debe crear un componente dinámico *SetMemberInfo*, cuya creación debe propagarse a cada réplica GMNS, donde se instancia la réplica de ese componente dinámico. El servidor GMNS que crea el grupo emplea un código que en esencia es:

```
SetMemberInfoImpl setImpl = new SetMemberInfoImpl(Consts.noInfo, domainHandler);
SetMemberInfo set = setImpl.theSetMemberInfo();
map.put(groupName, set);
set.add(new MemberInfo(groupHandler,memberId, clientReference));
```

- Se crea el componente empleando el constructor dinámico, que precisa de la información que se envía a las demás réplicas sobre el componente que se crea. En ese caso concreto no se envía ninguna información. Cuando este componente se crea, también lo hacen sus réplicas en los otros servidores GMNS.
- La segunda línea en el anterior listado es necesaria por la diferenciación CORBA entre servidor y *servant*[11] [OMG98]. Simplemente activa el servidor, obteniendo una referencia válida al mismo. En el caso de JavaRMI, *exporta* el servidor para ser públicamente accesible.

[11] En esencia, *servant* es la implementación en un lenguaje determinado de la funcionalidad del servidor. Es preciso activar (encarnar) el *servant* para obtener el servidor.

- Se inserta el *set* en el diccionario, con el nombre dado al grupo. Puesto que el componente diccionario está replicado, sus réplicas insertan simultáneamente el *set* con el mismo nombre. El *set* que insertan no es el existente en este servidor, sino el que haya sido creado en el respectivo servidor.
- Finalmente, se inserta la información del grupo en el *set*. De nuevo, esta operación se propaga a las demás réplicas, de acuerdo con la implementación de *SetMemberInfo*, con lo que la información es consistente en cada réplica.

Cada réplica del servicio GMNS debe ser capaz de crear componentes dinámicos, por lo que es necesario implementar la interfaz *DynamicSubgroupsUser*:

```
class MyDynamicSubgroupsUser extends DynamicSubgroupsUserBaseImpl
{
   public MyDynamicSubgroupsUser() throws Exception
   {
   }
   public GroupMember acceptSubgroup(int id, DynamicSubgroupInfo info)
   {
      try{return new SetMemberInfoImpl(id, domainHandler).theSetMemberInfo();}
      catch(Exception ex){return null;}
   }
   public GroupMember subgroupCreated(int creator, int id, DynamicSubgroupInfo info)
   {
      try{return new SetMemberInfoImpl(id, domainHandler).theSetMemberInfo();}
      catch(Exception ex){return null;}
   }
   public void subgroupRemoved(int remover, int id, DynamicSubgroupInfo info)
   {
   }
}
```

- Cuando se recibe cualquiera de los dos eventos de creación de componente, se crea un objeto *SetMemberInfo*. Puesto que es el único tipo de objeto que se crea, no hace falta que el parámetro *DynamicSubgroupInfo* contenga información específica.
- No es preciso hacer nada cuando el componente se elimina (una implementación C++ hubiera precisado la destrucción del *servant*).

Por último, la instancia de la anterior clase debe registrarse en el dominio, que debe crearse con las características deseadas. También deben registrarse en el dominio los componentes estáticos:

```
(A)   domainHandler = new DomainGroupHandlerImpl().theDomainGroupHandler();
(B)   domainHandler.setBehaviourMode(
         BehaviourOnViewChanges.MembersOnTransferExcludedFromGroup);
      domainHandler.setDynamicSubgroupsUser(
```

```
        new MyDynamicSubgroupsUser().theDynamicSubgroupsUser());
      domainHandler.setDomainGroupUser ( . . . );
(C)   GroupMonitorImpl monitorImpl =
        new GroupMonitorImpl(Consts.MONITOR_SUBGROUP, domainHandler);
      MapStringSetImpl mapImpl =
        new MapStringSetImpl(Consts.MAP_SUBGROUP, domainHandler);
      map = mapImpl.theMapStringSet();
(D)   monitorImpl.register();
      mapImpl.register();
```

- La única operación en el grupo A crea el dominio.
- Las siguientes tres líneas en el grupo B definen las características del dominio, incluyendo el registro del gestor de componentes dinámicos que permite emplear este tipo de componentes.
- A continuación, se crean los componentes estáticos, que en este caso es un monitor, empleado para las transacciones, y un diccionario.
- Finalmente, las dos operaciones del grupo D registran los componentes en el dominio.

11.3.3. Implementación de los algoritmos

Tras haberse implementado los componentes replicados y creado el dominio para gestionarlos dinámicamente, podemos centrarnos en la lógica de aplicación. La operación fundamental del servidor GMNS es la introducción de un servidor en un grupo, que tal vez sea necesario crear.

Para esta operación definimos dos operaciones básicas:

- *joinGroup*: incluye un miembro en un grupo. Si el grupo no existe, la operación devuelve *false*.
- *createGroup*: crea un grupo e inserta el miembro especificado. Si el grupo ya existe, la operación devuelve *false*.

Con este soporte, la operación *findAndJoinGroup* se escribe, sin soporte de errores, como:

```
. . .
boolean stop=true;
while(!stop) {
        stop=joiner.joinGroup(groupName, factory, memberId, reference);
        if (!stop) {
             stop=creator.createGroup(groupName, factory, memberId, reference);
        }
}
. . .
```

Primero se intenta extender el grupo; si no se consigue, se intenta crearlo pero, durante este intervalo, otro servidor puede haber creado el mismo grupo, con lo que la creación puede fallar también, en cuyo caso se repite el proceso. La alternativa a este algoritmo es emplear un monitor que bloquee el acceso al diccionario a otras réplicas, impidiendo que creen el mismo grupo. Sin embargo, la operación *join* es particularmente lenta, y esta aproximación supone bloquear a todos los servidores GMNS en cada operación, incluso aquellos que pretenden acceder a grupos diferentes.

La operación *createGroup* debe emplear transacciones para evitar que dos servidores creen el mismo grupo al mismo tiempo:

```
domainHandler.startTransaction(monitor);
SetMemberInfo group = map.get(groupName);
if (group==null) {
         ret=createGroupSafely(groupName, factory, memberId, reference);
}
else if (group.get()==null) {
        ret=addMemberSafely(group, factory, memberId, reference);
}
else {
  ret=false;
}
domainHandler.endTransaction();
```

- La creación del grupo se protege con una transacción en torno al monitor registrado.
- Se obtiene del diccionario la información asociada al grupo específico, que es una referencia nula si el grupo no existe.
- Si el grupo no existe, se crea. Si existe pero está vacío, simplemente se inserta el nuevo miembro (un grupo no se elimina automáticamente cuando queda vacío). Y si existe y tiene elementos, la operación falla y devuelve *false*.

La operación *joinGroup* no precisa de transacciones: un componente replicado admite accesos concurrentes sin problemas de consistencia, siempre que la semántica de la operación lo permita. Si dos réplicas insertan simultáneamente dos elementos en una lista, la semántica de esta operación implica que la lista termina con dos elementos. Pero si esta operación se realiza sobre un diccionario, y las dos réplicas emplean la misma clave, uno de los valores se pierde.

De esta manera, la operación más lenta, *join*, no bloquea a las réplicas. Esta operación obtiene la información del grupo y, si éste existe, solicita un miembro del grupo. Este miembro se emplea, a continuación, para incluir al nuevo miembro en su grupo. Si la operación tiene éxito, el nuevo miembro pasa a formar parte de ese grupo:

```
SetMemberInfo group=map.get(groupName);
if (group==null) {
        ret=false;
}
else {
        MemberInfo toJoin = group.get();
        if (toJoin == null) {
                ret = false;
        }
        else {
                ret = utilJoinGroup(toJoin.handler);
        }
}
```

Este código sólo muestra la lógica principal, puesto que el proceso de inclusión de un miembro en un grupo puede fallar por problemas de comunicaciones y el servidor GMNS debe entonces repetir el proceso, o devolver un error válido al cliente.

Tampoco se explican en este capítulo otros algoritmos de SenseiGMNS, por ser irrelevantes en cuanto a su aplicación de la metodología, incluyendo:

- Métodos para que el mismo servidor SenseiGMNS forme un grupo por sí mismo, publicando su información en puertos TCP específicos o en direcciones configurables.
- Control de referencias inválidas, pertenecientes a miembros excluidos de grupos. Sólo uno de los servidores GMNS realiza este control, pues no tiene sentido que todas las réplicas verifiquen las mismas referencias.
- Persistencia de la información, de tal manera que si todos los servidores GMNS se caen, no se pierdan los datos de los grupos existentes y sus miembros.
- Interfaz gráfica, que muestra los eventos producidos por los componentes mediante el patrón de observación.

11.4. Conclusiones

SenseiGMNS completa la funcionalidad del modelo de sincronía virtual no presente en SenseiGMS, permitiendo la creación y extensión de grupos. También presenta funcionalidad extendida para soportar la gestión de grupos mediante nombres.

La implementación es un ejemplo de empleo de la metodología soportada por SenseiDomains. La aplicación de esta metodología pretende facilitar el diseño de servidores replicados mediante el empleo de componentes, permitiendo que el

diseñador se enfoque en la lógica de la aplicación y no en los métodos de aplicar esa lógica.

Sin embargo, al mostrar cómo se ha diseñado SenseiGMNS, la implementación de su lógica de aplicación no ha sido inmediatamente posible. Primero, ha sido necesario desarrollar los componentes que la sustentan, tarea que conlleva un considerable esfuerzo. No obstante, debe tenerse en cuenta que, con el soporte de SenseiUMA, esos componentes habrían estado ya disponibles.

Pero incluso con el empleo de componentes replicados ya disponibles, es preciso que la aplicación realice una serie de pasos que permitan su posterior uso, tales como crear un dominio, registrar los componentes estáticos e instruir al dominio sobre cómo actuar con componentes dinámicos. Una vez concluidos estos pasos, es posible dedicarse a la lógica de la aplicación, empleando componentes replicados como si fueran *normales*, aunque compartidos por todas las réplicas.

El esfuerzo adicional para manejar el dominio es, sin embargo, muy inferior al necesario para programar una aplicación tolerante a fallos sin soporte de componentes. Procesos como la optimización que hemos realizado en el algoritmo para bloquear réplicas el mínimo tiempo posible son evidentemente factibles sin el empleo de componentes, diseñando, posiblemente, una lógica de aplicación muy diferente, pero la ventaja de esta aproximación es su semejanza a un caso no replicado y, consecuentemente, la posibilidad de emplear los mismos algoritmos. Esta lógica implica, además, una menor curva de aprendizaje, lo que resulta en una implementación más rápida para analistas no experimentados en aplicaciones tolerantes a fallos, así como en una más fácil migración de servidores no replicados a su equivalente replicado.